下载APP

08 | 抛出异常，是不是错误处理的第一选择？

2021-12-01 范学雷

《深入剖析Java新特性》 课程介绍 >

讲述：范学雷
时长 17:29 大小 16.03M

你好，我是范学雷。从今天开始，我们进入这个专栏的第二个部分。在这一部分，我们重点聊一聊代码的性能。这节课呢，我想跟你讨论 Java 的错误处理。

Java 的错误处理，算不上是特性。但是 Java 错误处理的缺陷和滥用，却一直是一个很有热度的话题。 其中，Java 异常的使用和处理，是滥用最严重，诟病最多，也是最难平衡的一个难题。

为了解决花样百出的 Java 错误处理问题，也有过各种各样的办法。然而，到目前为止，我们还没有看到能解决所有问题的好方法，这也是编程语言研究者们的努力方向。

不过也正是因此，我们就更需要掌握 Java 错误处理的机制，平衡使用各种解决办法，妥善处理好 Java 异常。我们还是通过案例和代码，来看看 Java 异常的滥用，以及可能的解决方案吧。

阅读案例

我们知道，Java 语言支持三种异常的状况：非正常异常（Error），运行时异常（Runtime Exception）和检查型异常（Checked Exception）。关于这三种异常状况的介绍，你可以参考《异常处理都有哪些陷阱？》这篇文章。

通常情况下，我们谈到异常的时候，除非有特别的声明，不然指的都是运行时异常或者检查型异常。

我们还知道，异常状况的处理会让代码的效率变低，所以我们**不应该使用异常机制来处理正常的状况**。一个流畅的业务，理想的情况是，在执行代码时没有任何异常发生。否则，业务执行的效率就会大打折扣。

异常处理对代码执行效率的影响有多大呢？我们先要对这个问题有一个直观的感受，然后才能体会“不应该使用异常机制来处理正常的状况”这句话的分量，认识到异常滥用的危害。

下面的这段代码，测试了两个简单用例的吞吐量。这两种状况，都试图截取一段字符串。但是其中一个基准测试没有抛出异常；另外一个基准测试，由于字符串访问越界，抛出了运行时异常。为了让两个基准测试更具有对比性，我们在两个基准测试里，使用了相同的代码结构。

复制代码

```
package co.ivi.jus.agility.former;

// snipped
public class OutOfBoundsBench {
    private static String s = "Hello, world!";  // s.length() == 13.

    // snipped

    @Benchmark
    public void withException() {
        try {
            s.substring(14);
```

```
        } catch (RuntimeException re) {
            // blank line, ignore the exception.
        }
    }

    @Benchmark
    public void noException() {
        try {
            s.substring(13);
        } catch (RuntimeException re) {
            // blank line, ignore the exception.
        }
    }
}
```

基准测试的结果可能会让你大吃一惊。没有抛出异常的用例，它能够支持的吞吐量要比抛出异常的用例大 1000 倍。

复制代码

```
Benchmark                        Mode   Cnt           Score            Error  Unit
OutOfBoundsBench.noException     thrpt   15   566348609.338 ± 22165278.114  ops/
OutOfBoundsBench.withException   thrpt   15      504193.920 ±    26489.992  ops/
```

如果用运营成本来衡量一下的话，你可以考虑按照使用的计算资源来计算费用的环境，比如云计算。如果没有抛出异常的用例要花一万块钱的话，抛出异常的用例就需要 1000 万才能支持相同数量的用户。如果一个黑客能够找到这样的运行效率问题，它足以让一个应用多掏 1000 倍的钱，或者直到应用耗尽分配的计算资源，无法继续提供服务为止。

这样的评估当然很粗陋，但是足以说明抛出异常对软件效率的影响。我们当然不希望我们编写的代码存在这么一个烧钱的问题。

这时候我们就会设想：我们的代码，能不能没有任何异常状况发生？我们前面也提到过，“一个流畅的业务，理想的情况是，在执行代码时没有任何异常状况发生”。

可惜，这几乎是无法完成的任务。随便翻一翻 Java 的代码，不管是 JDK 这样的核心类库，还是支持业务的应用软件，我们都能看到大量的异常处理代码。

比如说吧，我们要用 Java 搭建一个服务器。通常情况下，如果业务逻辑出现了问题，比如说用户输入的数据不合规范，我们都会抛出一个异常，标记出问题的数据，并且记录下来问题出现的路径。但是，无论出现什么样的业务问题，服务器崩溃都是不能接受的结果。所以，我们的服务器会捕获所有的异常，不管是运行时异常，还是检查型异常；然后从异常中恢复过来，继续提供服务。

但是场景是否异常有时候只是角度问题。比如说：输入数据不规范，从检查用户数据代码这个角度去看，这是一个不正常的情景，所以抛出异常；但是，如果从要求不间断运营的服务器的角度来看，这就只是一个需要应用程序妥善处理的正常状况，是一个正常的情景了。所以，服务器要能够从这样的异常中恢复过来，继续运行。

然而，现在稍微复杂一点的软件，都是很多类库集成的。大部分类库，都只从自己的角度考虑问题，并且使用异常来处理遇到的问题。除非是很简单的代码，不然我们很难期望一个业务执行下来没有任何异常状况发生。

毫无疑问，抛出异常影响了代码的运行效率。但是，我们又没有别的办法躲开这样的影响。所以，有些新的编程语言（比如 Go 语言）干脆就彻底抛弃了类似于 Java 这样的异常机制，重新拥抱 C 语言的错误码方式。

讨论案例

接下来的讨论，为了方便我们反复地修改代码，我会使用下面这个案例。

我们知道，在设计算法公开接口的时候，算法的敏捷性是必须要考虑的问题。因为，算法总是会演进，旧的算法会过时，新的算法会出现。一个应用程序，应该能够很方便地升级它的算法，自动地淘汰旧算法，采纳新算法，而不需要太大的改动，甚至不需要改动源代码。所以，算法的公开接口经常使用通用的参数和结构。

比如说，我们获取一个单项散列函数实例的时候，一般不会直接调用这个单项散列函数的构造函数。而是用一个类似于工厂模式的集成环境，来构造出这个单项散列函数的实例。

就像下面的这段代码里的 of 方法。这个 of 方法，使用了一个字符串作为输入参数。我们可以把它作为配置参数写在配置文件里。修改配置文件之后，不需要改动调用它的源代码就能升级算法了。

复制代码

```java
package co.ivi.jus.agility.former;

import java.security.NoSuchAlgorithmException;

public sealed abstract class Digest {
    private static final class SHA256 extends Digest {
        @Override
        byte[] digest(byte[] message) {
            // snipped
        }
    }

    private static final class SHA512 extends Digest {
        @Override
        byte[] digest(byte[] message) {
            // snipped
        }
    }

    public static Digest of(String algorithm) throws NoSuchAlgorithmException
        return switch (algorithm) {
            case "SHA-256" -> new SHA256();
            case "SHA-512" -> new SHA512();
            default -> throw new NoSuchAlgorithmException();
        };
    }

    abstract byte[] digest(byte[] message);
}
```

当然，通用参数也有它自己的问题。比方说，字符串的输入参数可能有疏漏，或者不是一个可以支持的算法。这时候，站在 of 方法的角度，就需要处理这样的异常状况。反映到代码上，of 方法要声明如何处理不合法的输入参数。上面的代码，使用的办法是抛出一个检查型异常。

那么，使用这个 of 方法的代码，就需要处理这个检查型异常。下面的代码，描述的就是一个使用这个方法的典型的例子。

复制代码

```java
try {
    Digest md = Digest.of(digestAlgorithm);
    md.digest("Hello, world!".getBytes());
} catch (NoSuchAlgorithmException nsae) {
    // snipped
```

```
}
```

既然使用了异常处理，当然也就会有我们在阅读案例里讨论过的异常处理的性能问题。我也试着给这个方法做了异常处理方面的基准测试。测试结果显示，没有抛出异常的用例，它能够支持的吞吐量要比抛出异常的用例大了将近 2000 倍。有了前面阅读案例的知识和铺垫，你应该对这样的性能差异早已有了心理准备。

复制代码

```
Benchmark                       Mode  Cnt           Score          Error  Units
ExceptionBench.noException     thrpt   15  1318854854.577 ± 14522418.634  ops/s
ExceptionBench.withException   thrpt   15      713057.511 ±     16631.048  ops/s
```

重回错误码

那么，既然异常处理的效率这么让人揪心，我们编写的 Java 代码能够像 Go 语言一样重回错误码方式吗？这是我们首先要探索的一个方向。

也就是说，如果一个方法不需要返回值，我们可以试着把它修改为返回错误码。这是一个很直观的修改方式。

复制代码

```
- // no return value
- public void doSomething();

+ // return an error code if run into problems, otherwise 0.
+ public int doSomething();
```

但是，如果一个方法需要一个返回值，我们就不能使用只返回错误码这种方式了。如果有一种方法，既能返回返回值，也能返回错误码，那么代码就会得到显著的改善。因此，我们需要设计一个数据结构，来支持这样的返回方式。

下面代码里的 Coded 这个档案类，就是一个能够满足这样要求的数据结构。

复制代码

```
public record Coded<T>(T returned, int errorCode) {
    // blank
};
```

如果一个方法执行成功，它的返回值应该存放在 Coded 的 returned 变量里；如果执行失败，失败的错误码应该存放在 Coded 的 errorCode 变量里。我们可以把讨论案例里的 of 方法，修改成使用错误码的形式，就像下面的这段代码这样。

复制代码

```
public static Coded<Digest> of(String algorithm) {
    return switch (algorithm) {
        case "SHA-256" -> new Coded(sha256, 0);
        case "SHA-512" -> new Coded(sha512, 0);
        default -> new Coded(null, -1);
    };
}
```

对应地，这个方法的使用就需要处理错误码。下面的代码，就是一个该怎么使用错误码的例子。

复制代码

```
Coded<Digest> coded = Digest.of("SHA-256");
if (coded.errorCode() != 0) {
    // snipped
} else {
    coded.returned().digest("Hello, world!".getBytes());
}
```

看了上面的代码，我想你应该已经能够判断出来它的性能状况了。我们还是用基准测试来验证一下我们猜想吧。

测试结果显示，没有返回错误码的用例，它能够支持的吞吐量和返回错误码的用例几乎没有差别。这就是我们想要的结果。

复制代码

```
Benchmark                   Mode  Cnt           Score          Error  Units
CodedBench.noErrorCode     thrpt   15  1320977784.955 ±  7487395.023  ops/s
CodedBench.withErrorCode   thrpt   15  1068513642.240 ± 69527558.874  ops/s
```

重回错误码的缺陷

不过，重回错误码的选择并不是没有代价的。刚才，我们在性能优化的同时，也放弃了代码的可读性和可维护性。异常处理能够解决掉的，也就是 C 语言时代的错误处理的缺陷，又重新回来了。

需要更多的代码

使用异常处理的代码，我们可以在一个 try-catch 语句块里包含多个方法的调用；每一个方法的调用都可以抛出异常。这样，由于异常的分层设计，所有的异常都是 Exception 的子类；我们也就可以一次性地处理多个方法抛出的异常了。

复制代码

```
try {
    doSomething();      // could throw Exception
    doSomethingElse();  // could throw RuntimeException
    socket.close();     // could throw IOException
} catch (Exception ex) {
    // handle the exception in one place.
}
```

如果使用了错误码的方式，每一个方法调用都要检查返回的错误码。一般情况下，同样的逻辑和接口结构，使用错误码的方式需要编写更多的代码。

对于简单的逻辑和语句，我们可以使用逻辑运算符合并多个语句。这种紧凑的方式，牺牲了代码的可读性，不是我们喜欢的编码风格。

复制代码

```
if (doSomething() != 0 &&
    doSomethingElse() != 0 &&
    socket.close() != 0) {
    // handle the exception
}
```

但是，对于复杂的逻辑和语句来说，紧凑的方式就行不通了。这时候，就需要一个独立的代码块来处理错误码。这样的话，结构重复的代码就会增加，这是我们在 C 语言编写的代码里经常见到的现象。

复制代码

```
if (doSomething() != 0) {
    // handle the exception
};

if (doSomethingElse() != 0) {
    // handle the exception
};

if  (socket.close() != 0) {
    // handle the exception
}
```

丢弃了调试信息

不过，重回错误码最大的代价，是可维护性大幅度降低。使用异常的代码，我们能够通过异常的调用堆栈，清楚地看到代码的执行轨迹，快速找到出问题的代码。这也是我们使用异常处理的主要动力之一。

复制代码

```
Exception in thread "main" java.security.NoSuchAlgorithmException: \
        Unsupported digest algorithm SHA-128
  at co.ivi.jus.agility.former.Digest.of(Digest.java:31)
  at co.ivi.jus.agility.former.NoCatchCase.main(NoCatchCase.java:12)
```

但是，使用错误码之后，就不再生成调用堆栈了。虽然这可以让资源的消耗减少，也能够提升代码性能，但是调用堆栈能带来的好处也就没有了。

另外，能够快速地找到代码的问题，也是一个编程语言的竞争力。如果我们决定重回错误码的处理方式，千万不要忘了提供快速排查问题的替代方案。比如使用更详尽的日志，或者使用启用 JFR（Java Flight Recorder）来收集诊断和分析数据。如果没有替代方案，我相信你会非常怀念使用异常的好处。

其实呀，C 语言时代的错误码，和 Java 语言时代的异常处理机制，就像是跷跷板的两端，一端是性能，一端是可维护性。在 Java 诞生的时候，有一个假设，就是计算能力会快速演进，所以性能的分量会有所下降，而可维护性的分量会放得很重。然而，如果演进到按照计算能力计费的时代，我们可能需要重新考量这两个指标各自所占的比重了。这时候，一部分代码可能就需要把性能的分量放得更重一些了。

易碎的数据结构

如果你阅读过我的另外一个专栏《代码精进之路》，你应该能够理解，一个新机制的设计，必须要简单、皮实。所谓的皮实，就是怎么用怎么对，纪律少、要求低，不容易犯错误。我们使用这样的准则，来看看上面设计的 Coded 这个档案类，是不是足够皮实。

生成一个 Coded 的实例，需要遵守两条纪律。第一条纪律是错误码的数值必须一致，0 代表没有错误，如果是其他的值表示出现了错误；第二条纪律是不能同时设置返回值和错误码。违反了任何一条纪律，都会出现不可预测的错误。

但是，这两条纪律需要编写代码的人自觉实现，编译器不会帮助我们检查错误。

比如下面的代码，对于编译器来说就是合法的代码。但对我们来说，这样的代码很明显违反了使用错误码需要遵守的规矩。这也就意味着，生成错误码的方式，不够皮实。

复制代码

```java
public static Coded<Digest> of(String algorithm) {
    return switch (algorithm) {
        // INCORRECT: set both error code and value.
        case "SHA-256" -> new Coded(sha256, -1);
        case "SHA-512" -> new Coded(sha512, 0);
        default -> new Coded(sha256, -1);
    };
}
```

我们再来看看使用错误码的代码。使用错误码，也有一条铁的纪律：必须首先检查错误码，然后才能使用返回值。同样，编译器也不会帮助我们检查违反纪律的错误。下面的代码，就没有正确使用错误码。我们需要依靠经验才能避免这样的错误。所以，使用错误码的方式，也不够皮实。

复制代码

```java
Coded<Digest> coded = Digest.of("SHA-256");
// INCORRECT: use returned value before checking error code.
coded.returned().digest("Hello, world!".getBytes());
```

需要的纪律越多，我们犯错的可能性就越大。那有没有改进的方案，能够减少这些额外的要求呢？

改进方案：共用错误码

我们希望，改进的方案能够同时考虑生成错误码和使用错误码两端的需求。下面这段代码就是一个改进的设计。

```
复制代码
public sealed interface Returned<T> {
    record ReturnValue<T>(T returnValue) implements Returned {
    }

    record ErrorCode(Integer errorCode) implements Returned {
    }
}
```

在这个改进的设计里，我们使用了封闭类。我们知道封闭类的子类是可以穷举的，这是这项改进需要的一个重要特点。我们把 Returned 的许可类（ReturnValue 和 ErrorCode）定义成档案类，分别表示返回值和错误代码。这样，我们就有了一个精简的方案。

下面这段代码，就是用新方案生成返回值和错误码的一个例子。可以看到，相比较使用 Coded 档案类的例子，这里的返回值和错误码分离开了。一个方法，返回的要么是返回值，要么是错误码，而不是同时返回两个值。这种方式，又把我们带回到了熟悉的编码方式。

```
复制代码
public static Returned<Digest> of(String algorithm) {
    return switch (algorithm) {
        case "SHA-256" -> new ReturnValue(new SHA256());
        case "SHA-512" -> new ReturnValue(new SHA512());
        case null, default -> new ErrorCode(-1);
    };
}
```

而且，生成 Coded 实例需要遵守的两条纪律，在这里也不需要了。因为，返回 ReturnValue 这个许可类，就表示没有错误；返回 ErrorCode 这个许可类，就表示出现错误。这样的设计，就变得简单、皮实多了。

接下来，我们再看看使用错误码的情况。下面的这段代码，我们使用了前面讨论过的 switch 匹配的新特性。Returned 这个封闭类被设计成了一个没有方法的接口，要想获得

返回值，我们就必须要使用它的许可类 ReturnValue，或者 ErrorCode。

复制代码

```
Returned<Digest> rt = Digest.of("SHA-256");
switch (rt) {
    case ReturnValue rv -> {
            Digest d = (Digest) rv.returnValue();
            d.digest("Hello, world!".getBytes());
        }
    case ErrorCode ec ->
            System.out.println("Failed to get instance of SHA-256");
}
```

如果一个方法的调用返回的是 Returned 实例，我们就知道，它要么是代表返回值的 ReturnValue 对象，要么是代表错误码的 ErrorCode 对象。而且，你要使用返回值，就必须检查它是不是一个 ReturnValue 的实例。这种情况下，使用 Coded 档案类编写代码需要遵守的纪律，也就是必须先检查错误码，在这里也不需要了。使用错误码的这一端，也变得更加简单、皮实了。

当然，使用封闭类来分别表示返回值和错误码的方式，只是改进错误码的其中一种方式。这种方式仍然具有一些缺陷，例如它本身没有携带调试信息。在 Java 的错误处理方面，我们希望未来能够有更好的设计和更多的探索，让我们的代码更完善。

总结

好，这节课就讲到这里，我来做个小结。从前面的讨论中，我们了解了 Java 异常处理带来的性能问题，我还给你展示了使用错误码的方式进行错误处理的方案。使用错误码的方式进行错误处理，错误码不能携带调试信息，这提高了错误处理的性能，但是增加了错误排查的困难，降低了代码的可维护性。

我们在代码里，是应该使用错误码，还是应该使用异常，这是一个需要根据应用场景认真权衡的问题。Java 的新特性，尤其是封闭类和档案类，为我们在 Java 的软件里使用错误码的形式，提供了强大的支持，让我们有了新的选择。

如果你想要丰富你的代码评审清单，错误码可以作为一个可评估的选项，进入你的考察指标内：

> 使用异常的机制进行错误处理，是不是一个最优的选择？

另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你们面试中出现哦。通过今天的学习，你应该能够：

清楚 Java 异常处理所带来的性能问题，对这一问题的影响程度有一个大致的概念；

- 面试问题：你知道 Java 异常处理会产生什么问题吗？

了解 Java 异常处理的替代方案，以及它的优势和劣势；

- 面试问题：你知道怎么提高 Java 代码的性能吗？

使用封闭类和档案类这样的 Java 新技术，为 Java 的错误处理寻求一个替代方案，这是一个崭新的、尚未开发的课题。在面试的时候，我们经常会遇到对代码性能有着苛刻要求的场景，如果你能够借助新特性展示错误处理的替代方案，并且不回避这个方案存在的问题，这一定是一个彰显你创新能力的好时机。

思考题

在前面的替代方案中，我们使用封闭类来分别表示了返回值和错误码，在使用错误码的代码里，我们使用了 switch 的模式匹配。可是，直到 JDK 17，switch 的模式匹配这个特性还只是一个预览版，还没有最终定稿。一般情况下，我们可以研究探索，但是不推荐使用预览版的特性。那么，如果不使用 switch 的模式匹配，使用错误码的代码可能是什么样子的呢？这是这一次的思考题。

为了方便你阅读，我把 switch 模式匹配的代码放在了下面。你可以在这个基础上替换掉 switch 模式匹配，看看最后会是什么样子的。

复制代码

```java
package co.ivi.jus.error.review.xuelei;

import co.ivi.jus.error.union.Digest;
import co.ivi.jus.error.union.Returned;

public class UseCase {
    public static void main(String[] args) {
        Returned<Digest> rt = Digest.of("SHA-256");
        switch (rt) {
            case Returned.ReturnValue rv -> {
```

```
                    Digest d = (Digest) rv.returnValue();
                    d.digest("Hello, world!".getBytes());
                }
            case Returned.ErrorCode ec ->
                    System.out.println("Failed to get instance of SHA-256");
        }
    }
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及验证的代码和结果。我们下节课再见！

注：本文使用的完整的代码可以从GitHub下载，你可以通过修改GitHub上review template代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在错误处理专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 error/review/xuelei 的目录下面。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | switch匹配：能不能适配不同的类型？

下一篇 09 | 异常恢复，付出的代价能不能少一点？

精选留言 (6)

写留言

郑晔

2021-12-03

这篇文章不算是 Java 新特性的介绍，而是基于 Java 新版本特性进行编码方式的一种探

索。

这种做法类似于 Rust 中提供的 Result。
https://doc.rust-lang.org/std/result/...
展开

作者回复: 没想到郑老师来看这篇文章， 并且留下了精彩的评论。 谢谢郑老师！

这个想法还有一些待完善的地方， 比如编译器怎么检验错误码； 比如模式匹配能不能改进，这样用户可以少一步调用。 看看更多时间的琢磨，有没有办法让它变得更简单、皮实。

6

Jxin
2021-12-01

1.这个用法眼前一亮。对于 rpc调用 返回 Result格式 的风格，采用这个写法不论调用方还是服务方，代码看起来都更干净整洁，可以有效提高可读性，提高知识传承的效率。
2.借助日志框架，你是能精细化到哪个类哪行代码打了这个日志。只要再把当前函数的关键入参追加在日志里，其实就足以发现并处理大部分问题了。需要堆栈的场景其实是为了还原真实调用的链路。假设入参组装有问题，就可以顺着这个链路往上翻找到可能出现...
展开

作者回复: 留言精彩啊！
2、也有很多场景，异常的性能开销无法忍受。
3、是的，错误码的缺陷之一。我期望的是下一讲里提到的（就一句话），Java异常处理自身的改进。目前能够看到的替代办法，都有或多或少的问题。
4、这个想法有意思！
5、参见3.

4

哦吼掉了
2021-12-01

小疑问：
Digest.of返回错误码的同时，新增一个属性把抛出异常的位置返回是否合适Thread.currentThread().getStackTrace()[1]。
思考题：可以用类型匹配，JDK16已正式发布。
if(rt instanceof ReturnValue rv && rv.returnValue instanceof Digest d){...
展开

作者回复: 返回异常的话，依然需要生成异常，不一定有多少性能的改进。

1

aoe
2021-12-02

性能优先的场景：多主动判断，少抛异常，只能辛苦程序员了
对性能没追求的场景：尽情抛异常，潇洒写代码，剩下的交给 JVM

作者回复: 哈哈，要是又潇洒又有性能，就理想了。

ABC
2021-12-01

https://github.com/XueleiFan/java-up/pull/15

展开

作者回复: 在PR里留言了，要是代码的规范在留意些，就更好了！

ABC
2021-12-01

其实错误码应用很广泛,也很方便.比如微信支付,支付宝支付等应用给第三方的接口文档,基本全都是错误码模式,返回格式是json.一个错误码,一个错误消息,一个消息主体,如果没有错误就返回0,错误消息为空.只是在使用的时候各不相同而已.

我记得,在Java里面抛出异常,会一直从触发异常的地方调用到栈顶,这是其中一个原因,是...

展开

作者回复: 异常的触发是这样的，这也就是我们说的堆栈信息。